CSC207 Design Document (Phase 2)

Ritik Kothari, Parth Amin, Petar Jovasevic, Azka Azmi, Meet Patel, Aryaman Modi

6 December 2021

Table of Contents

1 Updated Specification

1.1 Project Domain

Implement a fully functioning chess game (with additional features such as a leaderboard) with the correct logic and game rules. This game will be presented to the user using a clean and organized command-line GUI to facilitate an efficient user experience.

In Phase 0, our team created the chess board and also the 32 total pieces that are initialized. We followed the scenario walkthrough and programmed the first move, which covered a pawn moving either one or two spaces on the board.

In Phase 1, we expanded this to cover the entire course of a typical chess game, with only castling and a leaderboard system left to implement. Leaderboard features would include a system that keeps track of player moves until checkmate is achieved, with lower total player moves implying a higher position on the leaderboard. We will also consider a time tracker, in which the amount of time taken to achieve checkmate will be recorded, with high scores being dealt with in a similar fashion to the player moves.

Our project underwent a significant number of changes for Phase 2. Due to a lack of resources (mainly time), we decided against the implementation of a leaderboard and advanced chess moves such as en passant. We instead focused on refining our code. Extra care was taken to address the bugs from Phase 1, and to ensure the proper implementation of SOLID principles, Clean Architecture, and Universal Design principles. Proper Javadoc and other code etiquette was also followed during this Phase.

1.2 Project Domain Rules

The rules of our project domain remain unchanged from Phases 0 and 1. To reiterate, we list them below:

Chess is a two-player strategy game that takes place over an 8 x 8 board with one player controlling 16 black pieces and the other controlling 16 white pieces. Beginning with the player controlling the white pieces, each player makes one move per turn with the goal of capturing the opponent's King. The rules for each piece are as follows:

- Pawns: can move forward once (but may move forward twice on their first move of the game) and attack diagonally.
- Kings: can move and attack one block in any direction.
- Queens: can travel any number of blocks in any direction, attacking the block on which they land on.
- Bishops: can only move any number of blocks diagonally, attacking the block on which they land on.
- Rook: can move any number of blocks vertically and horizontally, attacking the block on which they land on.
- Knights: can travel two blocks horizontally and one block vertically (or vice versa), attacking the block on which they land.

White pieces are represented by lowercase names while black pieces are represented by uppercase names. Kings are represented using unit code. If a player's king is in a position to be attacked, then they are in check. If there are no possible moves the player in check can make to break the check, then they lose and therefore, the other player wins.

For Phase 2, we will consider implementing the en passant move, which involves the pawn piece. En passant is a special pawn capture that can only happen right after a pawn moves two squares from its starting position, in the instance where it could have been captured by an enemy pawn if it had advanced only one square. The opponent captures the just-moved pawn "as it passes" through the first square. Regardless of whether the pawn had advanced only one square or the enemy pawn had captured it normally, the result is the same. The en passant capture must be made on the very next turn or the capture is no longer valid.

Towards the end of Phase 2, we decided against the implementation of en passant, due to its obscurity and complicated implementation. Time was instead spent ensuring that the program was polished and contained no bugs.

2 Class Diagram

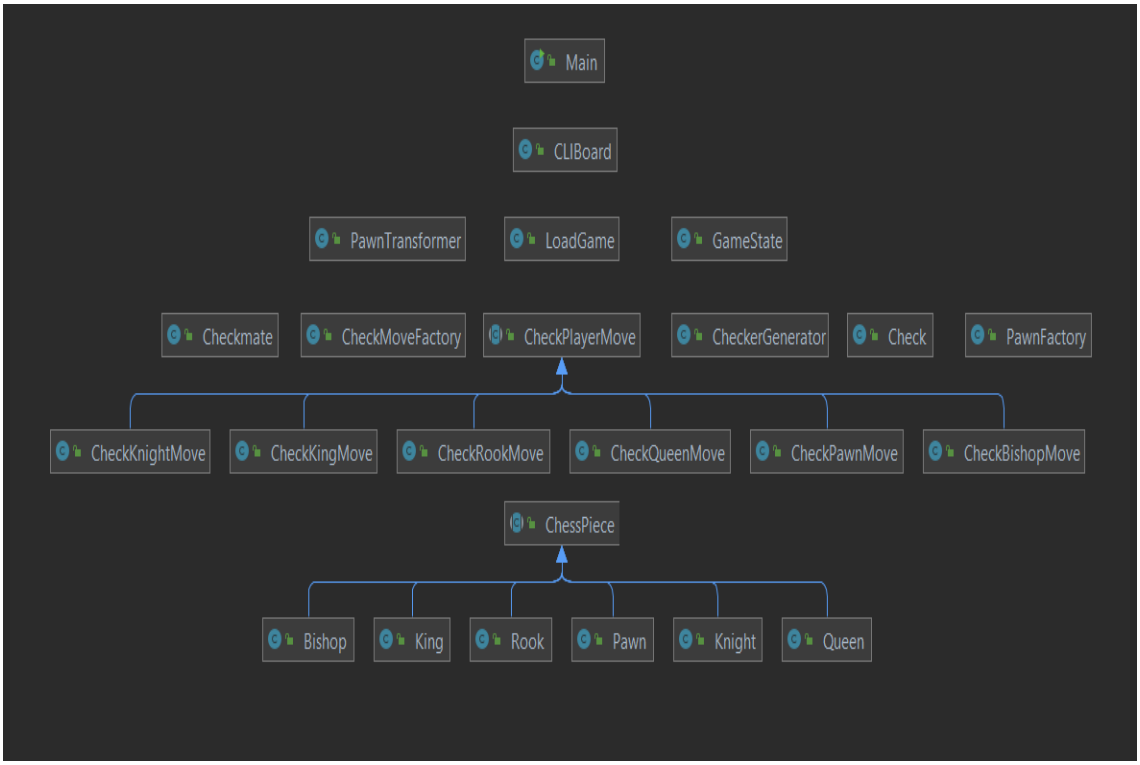

Figure 1: UML Class Diagram for Chess Game

Figure 1 shows the UML Class Diagram generated using IntelliJ Pro. It details the outline of our program, and shows the relations between each class. Clean Architecture is adhered to as seen in the diagram.

3 Design Decisions

3.1 Using Factory Methods

The first design decision we made was to consider using factory methods. We decided to use the same generic factory method to generate everything instead of using personalized ones, which saved time when it came to the implementation of other features.

We settled on two distinct methods, one for pawn and chess piece transformation and the other for checker generation.

3.2 Code Implementations

3.2.1 UI (User Interface)

Another major design decision we made was deciding how to implement the UI. We decided to leave the UI as is from Phase 0, which consisted of the command-line UI with letters representing the various pieces.

Our reasoning for this was that there were additional features that took precedence over the UI, and so we decided to devote more resources towards the implementation of those. The command-line UI is already an efficient method of displaying the state of the board, so we felt that there was no need to modify it.

No significant changes were made to the UI during Phase 2, as we consider it to be an effective method of displaying our program as it currently is.

3.2.2 Check and Checkmate

The final major design decision we made was figuring out how to implement the check and checkmate conditions.

During Phase 1, we came up with various ideas on how to implement Check. We considered the case where we determine if the King's location falls for any valid moves of an opposing piece, and also the case where we determine if there are opposing pieces in the King's path that can capture it.

We decided to implement the case where we determine if there are opposing pieces in the King's path that can capture it. This method was chosen because we saw it as more efficient compared to the other one, in that checkers must only be implemented for the King class rather than various pieces.

During Phase 2, we refactored the code for Check and Checkmate by first eliminating repetitive code. This was done by pulling repetitive code out and constructing it into a helper class called CheckMoveFactory. We then replaced all occurrences of instanceOf and

completely refactored Check to work based on a new algorithm. Check now "checks" to see if any valid moves of opposing pieces correspond with the King's position. Implementing this new algorithm helped remove much of the "hardcoding" from Phase 1, improving the overall quality of our program.

The Checkmate implementation works mostly the same as was demonstrated in Phase 1, with the only differences being a few alterations made to avoid future bugs.

Originally, we had it so that Check and Checkmate called the same method. In Phase 2, we instead created a new class that is called upon by both Check and Checkmate, which expands on our priority to reduce repetitive code. We also wrote code so that Checkmate calls a factory class to create instances of checkmate and checkplayer move.

3.2.3 Castling

For the implementation of Castling, a move involving the King and a Rook, we extended upon the class CheckKingMove. Namely, in the method which generates a list of valid moves, we created conditionals that verify whether the criteria for castling are met (King and Rook haven't moved, no pieces in between, King does not cross spaces that are under attack, etc.) and added the appropriate moves to the list that we return if the conditionals passed. Because this is recognized as a King move, our unchanged makeMove method sufficed to move the King across the board appropriately, but it did not handle the movement of the Rook object, which was uninvolved in this move from a coding standpoint. To remedy this, we added conditionals in makeMove that check whether the move being made was Castling (which is easily verifiable, as there are only two such moves), at which point we would create a variable to indicate this to the program and conditionals later in the method would move the Rook accordingly if necessary.

3.2.4 Pawn Transformation

Implementing the transformation of pawns when they reach the opposing side of the board from where they started, we managed to make proper use of pawnFactory. Once the conditions of a pawn transformation have been met, players can choose to transform a pawn into a Bishop, Queen, Rook, or Knight. We also created a new Presenter for this implementation.

3.2.5 Save State

A major focus of ours for Phase 2 was to correctly implement a working save game feature for our program. This would allow players to save their current chess game and come back to it at a later time. For this feature, we properly implemented serializable, which is an interface that converts an object into bytes in order to store it or transmit it to memory.

3.2.6 Undo Feature

Along with the save state feature previously mentioned, we also added an "undo" feature which lets players undo their most recent move, reverting the game state back to what it was prior to the move. The process for this addition included extending upon serializable to create deep clones that are then called on as needed. This implementation uses the Memento design pattern as it creates various stacks and pops them.

3.3 Refactor of CheckPlayer Classes

Phase 2 also saw the refactoring of the CheckPlayer Classes, specifically the refactoring of the Queen, Bishop, and Rook checkers. This was done to minimize "hardcoding" and repetitive code, while also making the code easier to read. Essentially, because the code used to generate a list of diagonal moves was repeated in the Queen and Bishop checkers while the code used to generate a list of vertical/horizontal moves was repeated in the Queen and Rook classes, we decided to make the code that generates a list of such moves into protected methods in the superclass of these checkers, CheckPlayerMove, allowing us to reduce the problem of generating moves for these three pieces to calling at most 2 methods from the superclass, thereby drastically reducing the length and complexity of these classes.

3.4 Extension of Main

A notable aspect of our code from Phase 1 was that if an invalid input was entered by a player, the program threw an error and prompted the player to try again. In Phase 2, we restructured this so that the program accepts any type of input from a player and continues to run, all while prompting the player to enter whatever the correct input may be.

We also added more personalized "game over" messages, that display which player won the game and also the number of moves in which they won the game. We also added messages which warn the user if his king is in check at the start of his turn.

4 Adherence to Clean Architecture

4.1 Adherence

Our project is divided into different layers while adhering to Clean Architecture.

We have the Entities layer, which are the various chess pieces, and as the inner core of our domain, they have no dependencies.

Next, we have the Use Cases, which is the application layer that manipulates the Entities. The Use Cases are responsible for determining the list of possible valid moves for the various

chess pieces. The Check and Checkmate classes are also a part of the Use Cases and they are responsible for determining whether the king is in a state of check or checkmate.

We then have the Controller layer, which interacts with our implementations in the previous layer, the Use Cases. Examples of classes that exist in the Controller layer are GameState and LoadGame.

We than have the UI layer, where we have the CLIBoard class which takes in the current Game state and creates and returns a chess board accordingly before and after every player's move.

Finally, we have the Main class which is responsible for running our entire program using the layers we discussed above.

4.2 Violations

In Phase 2, our group was able to correct all violations of Clean Architecture embedded within Phase 1. These previous violations namely consisted of the fact that several UseCase classes were importing the Controller GameState. These violations were fixed through altering the parameter requirements of many UseCases to instead take in instances of lower level entities.

5 Consistency with SOLID Design Principles

5.1 Consistency

We have implemented our project while adhering to the SOLID design principles. The different layers in which we have divided our project goes on to show the Single Responsibility Principle. Each class in our project performs a unique function, i.e. it only has one responsibility at hand. This has assisted in unit testing our code as we have generated unique tests to check each individual functionality of our project.

By enforcing the single responsibility principle and limiting the amount of hard-coding, we have made it so that our program can easily be extended upon in the future. This specific aspect of our program is what made implementing our new features (i.e castling, saveStave, Undo, etc.), and extending the scope and functionality of our program quite easy. In Phase 2 we were also able to strongly uphold the dependency inversion principle, as there is a clear level of abstraction between our higher level classes (i.e. Main, GameState, etc.) and lower level classes (i.e. ChessPiece, CheckPlayerMove). Furthermore, our implementation relies heavily on abstraction and polymorphism (namely within CheckPlayerMove and ChessPiece subclasses) in order to fill in neccesary details for algorithmic manipulation.

5.2 Violations

Through heavy enforcement of the Open-Closed principle in Phase 2, our group was able to fix many of the SOLID design violations from Phase 1, which namely included violations of the Dependency Inversion principle.

6 Packaging Strategies

Our project uses the packaging by layers strategy. We have separated our project into different layers in adherence to Clean Architecture.

- *Entities*: Bishop, King, Knight, Pawn, Queen and Rook, and their abstract parent class Chess Piece are all stored in the Entities Folder.
- *Use Cases*: The various move checkers for the different chess pieces are stored in the Use Cases Folder. We have the check and checkmate classes stored in the Use Cases folder along with our class named PawnFactory.
- *Utils*: The utils folder is a utility folder which stores a helper method for our gamestate class.
- *Controller*: Game state and Load game , our two controllers are present in the Controllers folder.
- *Presenter*: It stores the Pawn Transformer Class which is responsible for transforming a pawn into another chess piece when it reaches the end of the board.
- *User Interface*: Our User Interface is the Command Line Chess Board and the UI folder stores the CLIBoard class.

7 Design Patterns

7.1 Simple Factory

The Pawn Factory, ChessPiece factory and CheckerGenerator in the Use Case package utilizes the Simple Factory Pattern.

The Simple Factory Pattern describes a class that has one creation method with a large conditional that chooses which product class to instantiate and return based on method parameters.

The class Pawn Factory uses the Simple Factory pattern to return an instance of Chess Piece. It takes in a pawn, and the string literal representing the chess piece it is to be transformed into as parameters. It then returns the Chess Piece the pawn was to be transformed

into with the attributes of the pawn. For example, if the pawn is to be transformed into a rook, our method would return a rook with the attributes of the pawn.

ChessPiece Factory works in much of a similar way, expect where Pawn Factory was curated to work with the PawnTransformer gateway, Chess Piece factory is a more generic version that allows for the duplication of chess piece as well as the creation of other chess piece sub-classes based upon the parameters of another chess piece.

CheckerGenerator on the other hand, is a factory class that takes in a chess piece, and returns it's corresponding instance of CheckPlayerMove.

7.2 Memento

We considered implementing the Memento design pattern in the form of an undo/redo option for chess moves during Phase 1, but we decided to hold off on a possible implementation of this until Phase 2.

An undo/redo feature would take into account one of the major intents of Memento, which is to avoid the violation of encapsulation, so that capturing and externalizing an object's internal state allows it to be returned to this state later. In the instance of programming our chess game, the Controller layers would be modified to save a state of the game prior to a move being made. If a player then decides to withdraw their move, the previous state remains accessible to bring up as the current state of the game.

In Phase 2, we successfully managed to implement an undo/redo feature (see Section 3.2.6 for details). The implementation takes into account the nature of Memento, creating stacks of various game states that can then be called upon by the player. To avoid a violation of encapsulation, we utilize serializable to create a deep clone of the GameState class creating a separate instance with the same attributes and properties. Thus, we can push this clone class to our stack and keep track of past states of the game and when needed, revert the changes by setting the current state of the game to the previously cloned one.

8 Review of Progress

8.1 Summary of Updated Project Specification

The updated project specification describes the project domain, with progress during each phase outlined (see Section 1.1). It then outlined the rules of chess, which remained unchanged throughout Phase 0 and Phase 1 (see Section 1.2).

Expanding upon the project scenario walk-through from Phase 0, the project's progress has reached a point where any typical chess game can be played without exercising any advanced moves (i.e en passant).

As mentioned above, the program implements the features of a typical chess game. During Phase 2, work was done to implement the castling move, pawn transformation and a SaveState system that allows the user to continue to play their game where they left off, similar to taking a break in a real life chess game. (See Section 3.2)

8.2 Summary of Successful Design Features

To reiterate, some successful design features from Phase 0 include:

- Creating an abstract class called ChessPiece to model the behaviour and common functionalities of a chess piece
- Breaking down a large class into smaller classes to avoid over-complication and excessive responsibilities or functionality (adheres to the SOLID principles)
- Creating a string value that represents our chess game, and all 32 chess pieces. This was then printed out successfully into the command line as a UI

During Phase 1, additional features implemented include:

- Methods that cover the movement of every chess piece
- Methods to capture chess pieces
- Check and Checkmate states for the game
- The Simple Factory design pattern was followed for the creation of the Pawn Factory class

During Phase 2, additional features implemented include:

- Implementation of more advanced Chess game features (i.e. Castling, Pawn Transformation, etc)
- Implementation of a well functioning undo feature
- Implementation of a Save State feature that allows the user to save a state of their game and return to it later
- Refactoring, thorough debugging, the extension of UI features (i.e. variety of game over messages, warning of Checkmate features and other design quirks) and increased adherence to SOLID design and Clean Architecture.

(For further elaboration of Phase 2's added features, see Section 3.2).

8.3 Open Questions

Some open questions our group struggled with during Phase 0 include:

1. Choosing the most consistent implementation to handle the way pieces are moved such that it abides by the principles of Clean Architecture.
2. Had trouble deciding on the design of handling check and checkmate conditions (Ensuring no possible moves can be made to declare checkmate, preventing invalid moves when in check, etc.).
3. Had trouble deciding on potential extensions of the application like ELO score, history tracking and how that data can be efficiently stored and manipulated.
4. Determining the most efficient way to verify whether a given move is valid.

The above questions were largely answered during Phase 1, with our solutions outlined below:

1. We decided to implement Use Cases that determine the validity of moves and had our Controller class–GameState–make the necessary modifications.
2. For the design of handling check and checkmate, we decided on checking to see if there are pieces in the King's path that can capture it (see Section 3.3 for details).
3. We settled on the implementation of a leaderboard of least moves and least time taken to reach checkmate for possible extensions of the application. Specifics regarding this decision will be addressed during Phase 2.
4. To determine ways to verify the validity of a given move, we implemented several checkers that check a move's validity according to the rules outlined in Section 1.2.

As would be expected, the solutions of pending questions from Phase 0 invited the formation of several new questions that we hope will be answered in Phase 2. Some of those questions are listed below:

- Of the violations mentioned for the SOLID principles and Clean Architecture, which can be remedied without greatly altering the code and which cannot? For those that cannot, what parts of the code must be changed to avoid the violation?
- The rules surrounding the en passant move apply to an extremely specific situation. Implementation of en passant in our program would require the creation of multiple additional checkers for several conditions. For how obscure of a move en passant is, is it worth devoting resources towards its development that can otherwise be used for other aspects of the program?

- Concerning the addition of the leaderboard system, there are multiple factors that must be considered.
  - Our intent is to create a leaderboard system that carries over through many games so that the user can notice any trends in their play. How exactly would this be implemented?
  - We have also considered the deletion and addition of several users so that the leaderboard can keep track of multiple players. What would an outline of the addition and deletion feature look like?
  - Finally, we would require methods that keep track of the number of moves made and time taken to reach checkmate. What would these methods look like, and how would information from them be extracted so that they can be displayed in the leaderboard?

The majority of these questions were answered during Phase 2, specifically those involving en passant and the leaderboard system (see Section 1.1 and Section 1.2 for details).

To briefly summarize, we decided against the implementation of en passant and a leaderboard mainly due to time constraints and the urgent need to polish up other aspects of our code.

8.4 Summary of Group Member Roles and Next Steps

Ritik - Worked on the design document and added test cases for the Bishop, Knight, and Pawn classes. Also created an error looping system where the user is prompted to enter a valid chess move in the event of a non-integer input.

Link to Significant Pull Request: *Pull Request 33*
https://github.com/CSC207-UofT/course-project-group-34/pull/33

This pull request implemented the error checking system that prompts the user to provide valid chess moves. Its purpose is to avoid throwing errors when invalid inputs are given, and instead prompts the user to enter a valid input without breaking the program.

Parth - Worked on the main java class to add the ability to save the current state of the game and the load it across different runs of the program. Also added undo, redo and new game features and created a helper class to aid in the process. Also worked on the UX featues of the game to ensure a clear channel to play and make sure that inputs are valid and the game flows clearly. Also refactored packages to better align with design principles and make the repository more clean and coherent.

Link to Significant Pull Request: *Pull Request 26*
https://github.com/CSC207-UofT/course-project-group-34/pull/26

This pull request implemented the save and load game methods that allows users to store game states across different runs of the program. This pull request also implemented undo,

redo and new game features and expanded the main java class to better interact with the users and make the game process function more seamlessly.

Meet - Worked on implementing unit tests pertaining to "castling," a special move that occurs between the King and Rook piece. Additionally, tests involving attacking/capturing pieces and consequently updating the board correctly were also created. Worked on identifying and fixing errors in the program such as in the previously created unit tests involving the King piece.

Link to Significant Pull Request: *Pull Request 10*
https://github.com/CSC207-UofT/course-project-group-34/pull/10

This pull request implemented unit tests pertaining to the validity of moves for each chess piece, as well as tests that ensure that the program is able to correctly identity when a player is or is not in check. These tests ensure that the rules and logic of the chess game at the basic level remain fundamentally intact.

Petar - Implemented castling. Debugging. Refactored the Bishop, Queen, and Rook checkers to minimize repetitive code and improve readability. Created a new Presenter named PawnTransformer which, in conjunction with various modifications to Main and GameState, ultimately allowed pawns to transform when they reach the end of the board.

Link to Significant Pull Request: *Pull Request 11*
https://github.com/CSC207-UofT/course-project-group-34/pull/11

This pull request saw the implementation of castling into the flow of the program, significant modifications to the packaging structure, and changes to the code that improved our program's adherence to Clean Architecture.

Azka - Worked on refactoring Check and Checkmate, changes to GameState, added more factory classes (i.e. ChessPieceFactory), improved upon current factory classes, improving the overall extensibility of our code, debugging as well as initial implementations of the Undo feature. Also contributed to code related sections within the design document.

Link to Significant Pull Request: *Pull Request 19*
https://github.com/CSC207-UofT/course-project-group-34/pull/19

This pull request contributed a significant amount to our program as it fixed quite a few SOLID and Clean architecture violations, improved the extensibility of our code and demonstrated a great detail of refactoring.

Aryaman - Worked on the design document, added methods to display end of game messages, and worked on debugging our code. Also added methods to inform the player whether their king is in check at the start of their turn, and to display the number of moves made by the winning player.

Link to Significant Pull Request: *Pull Request 15*
https://github.com/CSC207-UofT/course-project-group-34/pull/15

This pull request added the end of game messages for our chess game. After checkmate occurs, it displays which player won, and also displays the number of moves in which they won the game.

9 Project Accessibility Report

9.1 Principles of Universal Design

All answers are provided while referencing the contents of https://universaldesign.ie/what-is-universal-design/the-7-principles/#p5

9.1.1 Equitable Use

Considering the multiple guidelines of the Equitable Use design principle, our program demonstrates the following:

- Provides access through the same means to all users, in the form of a command line UI and general commands used in chess.
- Avoids segregating or stigmatizing users, by similar reasoning to what is mentioned above.
- Provisions for privacy, security, and safety are available to all users. As an offline program, privacy, security and safety measures are dependent upon on the measures provided by the user's device.

Our use of a command line UI might not be appealing to all users. However, due to the simplicity of a command line UI, we feel that an option catering to every individual's preferences is not required as the program's appearance does not take away from its other aspects.

9.1.2 Flexibility in Use

Considering the multiple guidelines of the Flexibility in Use design principle, our program demonstrates the following:

- Accommodates left/right-handed access and use. Since a keyboard is the main peripheral used to play our chess game, the limitations of the player's equipment are strictly in line with the accessibility limitations of our program.
- Facilitates the user's accuracy and precision. Again, as a keyboard is the primary peripheral used to operate our program, the user's accuracy and precision is dependent on the limitations of the user's hardware.
- Provides adaptability to the user's pace. This particular guideline is implemented through the undo/redo and save state features, allowing users to save their progress while also being able to undo accidental moves.

As a chess game programmed to display information in command line UI form, our program does not have much choice in methods of use. Possible features that could be implemented in the future to adhere to this guideline include porting over our program to Android Studio, so that users can access the program using any Android-supporting device.

9.1.3 Simple and Intuitive Use

Considering the multiple guidelines of the Simple and Intuitive Use design principle, our program demonstrates the following:

- Eliminates unnecessary complexity. Our primary decision to avoid complexity was to implement a command line UI, which displays information in a simple and efficient manner.
- Is consistent with user expectations and intuition. As our program is a chess game, users familiar with chess will find that our program adheres to the typical expectations one has while playing chess.
- Accommodates a wide range of literacy and language skills. While our program displays information entirely in English, the use of symbols and images break the language barrier to allow users to play despite whatever their English competency may be.
- Arranges information consistent with its importance. The program displays information as a chess game is played, with relevant information being shown during its respective stage in the game.
- Provides effective prompting and feedback during and after task completion. As mentioned in Section 3.4, our program prompts the user when invalid moves are made and also provides messages for "game over" and "Check" situations.

9.1.4 Perceptible Information

Considering the multiple guidelines of the Perceptible Information design principle, our program demonstrates the following:

- Maximizes "legibility" of essential information. As all information presented by our program is essential, the command line UI presents everything efficiently.
- Differentiates elements in a way that can be described. When the user is prompted to enter a valid move or is made aware of a Check, this information is provided in the terminal part of the application and is separate from the game itself.

- Provides compatibility with a variety of techniques or devices used by people with sensory limitations. Those with said sensory limitations will likely have access to hardware/software that enables them to access other applications easily. As our program is accessible by Java IDEs, any such peripheral compatible with IntelliJ or other compilers will also be compatible with our program.

As mentioned earlier, our program uses a command line UI, which does not display information in different modes (everything is text-based). A possible feature that could display information in different forms would again be to create an android app that displays information in pictorial or verbal form. This implementation can also provide adequate contrast between essential information and its surroundings.

9.1.5 Tolerance for Error

Considering the multiple guidelines of the Tolerance for Error design principle, our program demonstrates the following:

- Arranges elements to minimize hazards and errors. As everything is displayed in a clean command line UI format, information is also presented in a way that minimizes errors (tells the user how to move pieces in a way that is accepted by the program from the start).
- Provides warnings of hazards and errors. As mentioned previously, our program prompts the user when an invalid chess move is made, along with providing prompts for Check and Checkmate.
- Provides fail-safe features. In the event of an invalid input that would force the program to crash, the program instead prompts the user to re-enter inputs until a valid input is read.

There are not any tasks that require vigilance in our program. As a result, there is no need to discourage unconscious action for said tasks.

9.1.6 Low Physical Effort

Considering the multiple guidelines of the Low Physical Effort design principle, our program demonstrates the following:

- Allows the user to maintain a neutral body position. Since our chess game is accessible through a Java compiler available on a computer, the user would not need to keep changing their position in order to play chess.

- Uses reasonable operating forces. The physical extent of our program's requirements are simply to type move commands using a keyboard or other peripheral.
- Minimizes sustained physical effort. As regular chess is not a game which requires one to constantly move pieces around (our program does not force the user to play speed chess), there is no need for any sustained physical effort.

Due to the nature of chess, repetitive actions are essential to reach the end of a chess game. In a digital form such as in our program, users will have to enter chess moves repeatedly until their session is over. There are no possible implementations which could reasonably alter this, as that would mean changing what chess is about.

9.1.7 Size and Space for Approach and Use

Considering the multiple guidelines of the Size and Space for Approach and Use design principle, our program demonstrates the following:

- Provides a clear line of sight to important elements for any seated or standing user. Again, this is highly dependent on the user's hardware, so a well ergonomically designed work space should pose no problem in playing our chess game.
- Makes reach to all components comfortable for any seated or standing user. This holds with similar reasoning to what is mentioned above.
- Accommodates variations in hand and grip size. This is again largely dependent on the user's hardware, our program is accessible by many different peripherals.
- Provides adequate space for the use of assistive devices or personal assistance. Holds due to similar reasoning as for the above.

9.2 Target Audience of Program

When considering the target audience of our chess program, we would settle on people who like games. Given that our program is a faithful recreation of a classic chess game, people with varying levels of chess abilities would be able to enjoy playing our game. As our design combines simplicity with efficiency, it is suitable for players of all ages and does not possess a particularly steep learning curve. As such, we would expect that our program would sell well if it was marketed as a chess application to those who enjoy playing games.

9.3 Probability of Use by Demographics

Despite our program being a recreation of a timeless game, when considering the digital nature of our rendition, we could expect older demographics to be less likely to use our program. A stereotype surrounding older people is that they are less in tune with technology than the younger generations. While mostly true for complicated pieces of software/hardware, our chess program aims to be as simple as possible. Once introduced to properly, we believe the older generation would have the same experience playing our chess game as the younger generation would. However, the methods to access our program might off put those who are not familiar with computers, even those who enjoy games. It is due to these reasons that we would expect members of the older generations to be less likely to use our program.